Project Report: Reinforcement Learning for Circuit Synthesis from Frequency Response

1. Introduction

In analog electronics, there are numerous instances where the frequency response of a circuit is available, but the actual circuit topology and parameters are unknown. The core challenge lies in **modeling the circuit** based solely on its frequency response, as the underlying circuit that produced the response is inaccessible.

A motivating analogy is the frequency response of the **human body**—we can measure the response but can only hypothesize about the equivalent circuit representation. Currently, this task is predominantly tackled using **trial-and-error methods**. There is no established algorithmic approach to systematically synthesize a circuit from a given frequency response graph.

In this project, we propose a **Reinforcement Learning (RL)** based framework by formulating the problem as a **Markov Decision Process (MDP)**. Our objective is to leverage RL to learn an **optimal policy** that can synthesize a circuit matching a target frequency response.

---

2. MDP Formulation

A Markov Decision Process (MDP) is characterized by four components:

- **State Space (S)**
- **Action Space (A)**
- **Reward Function (R)**
- **Transition Probability Matrix (P)**

2.1 State Space (S)

The state space is divided into two key components:

a. Current Circuit Configuration

The current circuit is described by the set of passive components and their attributes:

- **Component Type (comp_type)**: {Resistor (R), Capacitor (C), Inductor (L)}
- **Component Value (comp_value)**: Continuous value within a specified range
- **Node1 (n1)**: Starting node of the component connection
- **Node2 (n2)**: Ending node of the component connection

This results in a tuple:
(comp_type, comp_value, node1, node2)

b. Target Frequency Response

To guide the RL agent towards matching the desired response, we encode the target graph directly into the state:

- The frequency response graph is divided into **10 segments**.
- For each segment, we compute:
    - **Slope** of the segment
    - **Mean value** of the segment

These features collectively represent the **Target Graph Descriptor**, providing necessary information for the RL agent to align the synthesized circuit's response with the target.

Thus, the complete state space **S** is a combination of:

- Current Circuit Configuration
- Target Graph Descriptor

2.2 Action Space (A)

The action space defines how the agent can modify the circuit. Each action specifies:

- **Component Type (comp_type)**: {R, C, L}
- **Component Value (comp_value)**: Within allowable range
- **Node1 (n1)**: Connection start node

- **Node2 (n2)**: Connection end node

This is represented as a tuple of length 4:
(comp_type, comp_value, node1, node2)

Essentially, the agent decides **which component** of **what value** to **place where** in the circuit.

2.3 Reward Function (R)

The reward function is designed to:

- **Encourage circuits** whose frequency response closely matches the target
- **Penalize unnecessarily complex circuits** with excessive components

The reward is computed as follows:

- **Correlation-based Reward**:
  We calculate the correlation between the **current circuit's frequency response** and the **target graph**.
  - If the correlation is **≥ 80%**, the goal is considered achieved, and a **high positive reward** is given (e.g., **+100**).
  - If the correlation is **< 80%**, a **negative reward of -1** is assigned.
- **Component Penalty**:
  For **each additional component** added to the circuit, a **negative reward of -1** is applied.
  This ensures that the optimal policy seeks to minimize both the **error** and the **circuit complexity**.

Overall, the reward function incentivizes the agent to synthesize the target response **efficiently**, using the **least number of components**.

2.4 Transition Probability Matrix (P)

Given the **deterministic** nature of the problem:

- At a state **s**, taking action **a** will deterministically transition the system to a new state **s′**.

- The transition probability **P(s′ | s, a) = 1** for the specific resultant state, and 0 otherwise.

Thus, the MDP transitions can be fully described without uncertainty.

---

3. Objective

The ultimate goal is to train the RL agent to discover a sequence of actions (i.e., a circuit topology and parameter selection) that **minimizes the discrepancy** between the synthesized circuit's frequency response and the target graph.

---

4. Why We Used the PPO Algorithm

For this project, we adopted the **Proximal Policy Optimization (PPO)** algorithm due to its suitability for complex, high-dimensional decision-making tasks like circuit synthesis.

Designing a circuit involves making **multiple interdependent decisions** simultaneously — selecting component types, choosing their precise values, and determining how they are connected within the circuit topology. These decisions involve a **combination of discrete choices** (e.g., component type, node connections) and **continuous parameters** (e.g., component values), making the action space rich and intricate.

PPO is particularly effective in such scenarios for several reasons:

- **Handles complex action spaces:** PPO efficiently manages environments where actions are not simple binary decisions but a **blend of numerical and categorical choices**, aligning well with our circuit design task.

- **Stable learning updates:** PPO improves the agent's policy incrementally by **constraining updates** (using clipped objective functions). This prevents the learning process from making overly large, unstable changes, ensuring **consistent progress** and avoiding convergence to poor solutions early on.

- **Balanced exploration and exploitation:** PPO naturally balances **exploration** (trying new actions) and **exploitation** (refining known good strategies), enabling the agent to discover **better solutions** over time while maintaining a focus on the target objective.

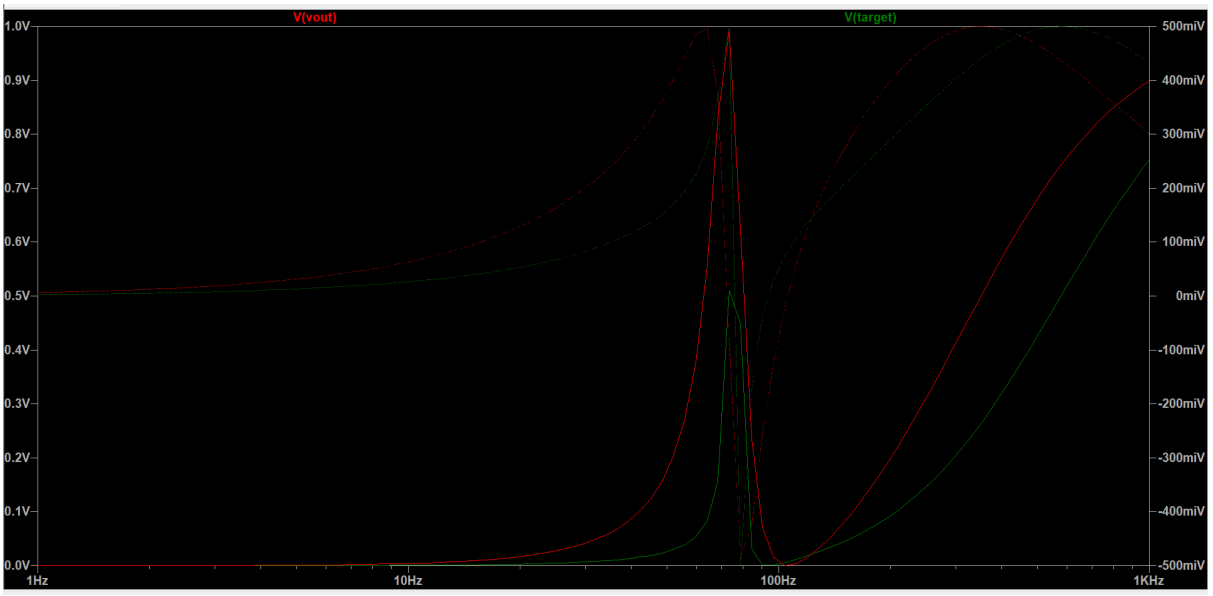
- **Ease of implementation and tuning:** Compared to older policy gradient methods, PPO is **simpler to implement** and requires fewer hyperparameter adjustments, making it more practical for iterative experimentation.

These attributes make PPO a **robust and effective choice** for training agents in our circuit synthesis task, where **meeting intricate design goals** with efficiency and reliability is critical.

5. Results and Analysis

The results of our Reinforcement Learning-based circuit synthesis approach are illustrated in **Figure 1** below. In the figure:

- The **red graph** corresponds to the **frequency response** of the circuit synthesized by our RL algorithm.
- The **green graph** represents the **target frequency response** that we aimed to replicate.



Although the **overall trends** and **general shape** of the two graphs align well, it is evident that the **exact range of values** does not match perfectly. This discrepancy arises primarily due to two critical factors — **limitations in the state space representation** and the **reward function formulation**.

5.1 Limitations in State Space Representation

In our state space design, we divided the **target frequency response** into **10 segments**, extracting:

- The **slope** of each segment (10 values)

- The **mean value** of each segment (10 values)

This resulted in a **20-dimensional vector** (10 slopes + 10 mean values) to represent the entire target graph.

However, the **original frequency response graph** inherently contains **over 5000 dimensions** (due to high-resolution sampling). Reducing this rich, high-dimensional information into a compact 20-dimensional vector leads to **information loss**, meaning that many subtle but significant features of the target graph might not be adequately captured.

This dimensionality reduction was intentionally chosen to **ensure faster convergence** of the RL model during training. However, for higher accuracy and better fidelity to the target graph, future improvements could include:

- **Increasing the number of segments** (i.e., using a higher-dimensional representation)
- Incorporating additional graph features such as:
    - **Minimum and maximum values**
    - **Locations and values of local maxima and minima**
    - **Overall energy or area under the curve**

These enhancements are expected to provide a **richer state representation**, helping the agent make more informed decisions.

5.2 Impact of Reward Function Formulation

The second, and arguably **more significant**, factor affecting the result is the **formulation of the reward function**.

Our current reward function assigns:

- A **high positive reward** if the **correlation** between the synthesized response and target response exceeds **80%**
- A **penalty of -1** if the correlation falls below this threshold
- An additional **-1 penalty** for every extra component added to the circuit to encourage **simpler designs**

While this strategy successfully ensures that the agent **matches trends** and achieves a **high correlation (>80%)**, it does not guarantee **pointwise accuracy** between the two

graphs. As a result, although the two responses correlate strongly in shape, the **exact magnitudes** may differ.

We hypothesize that **refining the reward metric** — by incorporating additional similarity measures (e.g., **Mean Squared Error (MSE)** between graphs, frequency-weighted errors, etc.) — could guide the agent toward producing **more precise and accurate circuit responses**.

6.Environment formulation:

```python
class CircuitEnv(gym.Env):  # Custom Environment


    def __init__(self, max_components=12, value_buckets=10):

        # print("Init")
        super(CircuitEnv, self).__init__()

        self.max_components = max_components  # Maximum number of components possible in a circuit.
        self.value_buckets = value_buckets  # Maximum number of possible values each compomnent can have.
        self.node_counter = 2  # 0 = GND, 1 = VDD --> Each circuit starts with 2 nodes (GND and VDD)
        self.episode_number = 0
        self.freq_points = 10
        self.corr = 0

        # You can define custom buckets later for R, L, C
        self.bucket_ranges = {
            0: np.logspace(0, 6, self.value_buckets),     # R: 1Ω to 1MΩ
            1: np.logspace(-6, -3, self.value_buckets),   # L: 1uH to 1mH
            2: np.logspace(-12, -6, self.value_buckets)   # C: 1pF to 1uF
        }

        self.component_list = ["res", "cap", "ind"]

        # Action = [component_type, value_index, node1, node2]
        # 0 --> R, 1 --> L, 2 --> C  component_type
        # Each value_bucket contains a list of values for each component. The action space will specify the index of the v
        # The next two vlaues are the node indices (possible values where we can put a component)
        self.action_space = spaces.MultiDiscrete([3, value_buckets, max_components + 2, max_components + 2])

        # Observation: padded list of components: [type_id, value_idx, node1, node2]
        self.observation_space = spaces.Dict({
            "components": spaces.Box(-1, 1, shape=(self.max_components, 4), dtype=np.float32),
            "target_response": spaces.Box(0, 1, shape=(2, 10), dtype=np.float32),
            })

        # Creating the train data object
        folder_path = "train_graphs/csvs"
        self.processor = rd.CSVProcessor(folder_path)

        self.LTSpice = "C:/Users/jishu/AppData/Local/Programs/ADI/LTspice/LTspice.exe"

        self.reset()
        # print("Init Done")
```

The environment is called CIrcuitEnv. During initialization, we pass two values (max_components and value_buckets). Max_components mean the maximum number of components a circuit can have. Value_bucket is the bucket length of values for each component. value_buckets=10 implies we have 10 values for resistor, 10 values for capacitor and 10 for inductor.

Self.processor is basically used for reading the target graphs so that we can include the target information in the state space.

We have defined self.action_space as [comp_type, comp_val, node1, node2]

We also defined the observation space to contain 2 values:

1. Components: This is a list of values which represent the current circuit. Everytime an action is taken, we add a list [comp_type, comp_val, node1, node2] to the components list. Based on this list, we form the circuit in the LTSpice simulator.
2. Target_response: This is to include the important information of target response in a lower dimensional vector. Every graph in the form of csv is read and is divided into 10 segments.. We compute and store the average and slope of each segment in hopes that it might capture the important information of the target response. So it can figure out the optimal policy.

```
def reset(self, *, seed=None, options=None):
    super().reset(seed=seed)

    self.components = []  # List of components as tuples (type, value_idx, n1, n2)
    self.G = nx.DiGraph()
    self.G.add_nodes_from([0, 1])  # 0 = GND, 1 = VDD
    self.node_counter = 2


    self.norm_avgs, self.norm_slopes, self.target = self.processor.process_next()
    self.current_file = self.processor.files[self.processor.index - 1]

    self.episode_number += 1

    obs = self.get_observation()

    return obs, {}
```

This is the function to reset the circuit after every episode. Each episode ends when either it reaches the max_components or the correlation of the current graph and required graph is more than 80%. We reset the component list, make the node_counter=2 which means the circuit contains only VDD and GND.

```python
def step(self, action):  # This method is called everytime the agent takes an action

    # print("In Step")

    new_node_flag = False # This tells if a new node is added in the circuit
    old_node = None # This keeps track of the old node value which now becomes the new node. Useful for updating the graph of nodes.
    # Action unpacking
    comp_type, val_idx, node1, node2 = action

    # Is the action valid in the current state or not?
    # This ensures that that there are no floating nodes in the circuit

    valid_nodes = len(self.G.nodes) # This gives me the current number of nodes in the circuit as well as the value of the next node.
    # print(f"Valid Node ={valid_nodes}")
    # You can define: current_node_limit = len(valid_nodes)
    if node1 > valid_nodes or node2 > (valid_nodes - 1):
        # Invalid node selection
        return self.get_observation(), -5, False, False, {}

    # If both nodes are same, that means one extra node is required to be added
    if node1 == node2:

        old_node = node1
        node1 = valid_nodes
        new_node_flag = True

    # value = self.get_value(comp_type, val_idx)  # Based on the val_idx and the comp_type, extract the value of the component

    # Handle new node assignment
    for node in [node1, node2]:
        if node not in self.G.nodes:
            self.G.add_node(node)  # Add the node to the graph representing the circuit

    # print("Adding")
    # Add component to circuit


    # Check if new_node is there. If it is we need to update the entire graph
    if new_node_flag == True:
        # print("Updating Graph")
        self.G = self.update_Graph(self.G, node1, old_node)

    self.components.append((comp_type, val_idx, node1, node2))

    # print("Adding Edge")

    self.G.add_edge(node1, node2, type=comp_type, value=val_idx)



    # Running the simulation and getting the result from LT Spice
    components = []
    gain = []
    # print(self.bucket_ranges)

    for item in self.components:
        # print(self.component_list.index(item[0]))
        # print(self.get_value((item[0]), item[1]))
        components.append([self.component_list[item[0]], self.get_value((item[0]), item[1]), item[2], item[3]])

    if len(self.G.nodes) > 2:
        self.build_circuit(components, "custom_rl_circuit.asc")
        self.run_simulation("custom_rl_circuit.asc", self.LTSpice)
        gain = self.plot_response("custom_rl_circuit.raw", "target.csv")


    obs = self.get_observation()
    reward = self.get_reward(gain, self.target)
    done = self.is_done()

    return obs, reward, done, False, {}
```

This is the action space. Every time it generates a list [comp_type, comp_val, node1, node2]. The list is handled appropriately in this function so that we can generate a circuit based on this in LT Spice.

```python
def get_observation(self):
    obs = np.full((self.max_components, 4), -1, dtype=np.float32)

    for i, (ctype, v_idx, n1, n2) in enumerate(self.components[-self.max_components:]):
        value = self.get_value(ctype, v_idx)  # Based on the val_idx and the comp_type, extract the value
        obs[i] = [
            ctype / 2,              # Normalize type (0,1,2)
            value / (self.value_buckets - 1),  # Normalize value index
            n1 / (self.max_components + 1),
            n2 / (self.max_components + 1)
        ]
    target = np.stack(
        [self.norm_avgs, self.norm_slopes], axis=0
    ).astype(np.float32)
    return {
        "components": obs,
        "target_response": target
    }
```

Get_observation returns 2 things: components and target_response. The format for these two have already been discussed above.

```python
def get_reward(self, gain, target):

    if len(gain) == 0:
        return -1
    gain1 = np.array(gain).reshape(1, -1)  # Reshape to 2D
    target1 = np.array(target).reshape(1, -1)  # Reshape to 2D
    target1 = np.array(target1, dtype=np.float64)

    self.corr = np.corrcoef(gain1, target1)[0, 1]
    reward =-len(self.components)
    if self.corr >= 0.8:
        reward = 10
    return reward

def is_done(self):
    return self.corr >= 0.8
    return len(self.components) >= self.max_components
```

Reward is +10 if correlation is more than 80%. Else it is -1 reward. For every component added also we get a -1 reward.

Random Circuit Generation for Training:

To train the agent we have generated circuits by randomly selecting the number of components between as a number between 2 and 15. Generated circuits are simulated to get the frequency response by simulating with LTSpice and the frequency response is obtained is saved in CSV files.

While simulation if the LTSpice is not able to simulate the circuit and throw an error when there are any floating nodes or absence VIn and VOut, the circuit is discarded.After removing the circuits that are invalid, the number circuits that are valid are 1895, these 1895 circuits are used for training the agent.

Circuit Building:

We have written a script that takes the components list as input and generates the LTSpice executable file in .asc format.

The input is list of components in this format {componenet_type, componenet value, node1, node2}

Simulation:

We execute the LTSpice file which is .asc format using

subprocess.run([ltspice_path, "-b", "-run", circuit_path], check=True, capture_output=True, text=True)

ltspice_path: where the executable "LTspice.exe" located,

circuit_path: where the .asc file generated is located.

This simulation generated a raw file which is then we read and load the frequency and gain.

These are used for either reward calculation or we save as a training for RL agent.

For Testing:

1. Install LTSpice and the required libraries

   gymnasium, networkx, numpy, stable_baselines3 etc

2. Change the path to the LTSpice executable in the CustomEnv.py file.

3. In the "test_graphs" folder the csv of thr required frequency response should be placed.

4. Run the code in the "test.py" file.

For Training:

1. Install the above mentioned libraries, and setup paths

2. Use the "buildckt_simulate.py" to create CSV files with frequency responses of circuits that will be generated randomly.

3. Either use the already existing CSV files in the "train_graphs" folder or generate new ones by following the step2

4. Run the code in the "train.py" file.

---

7. Conclusion and Future Work

This project introduces a novel RL-based approach to **circuit synthesis from frequency response graphs**—a task traditionally tackled by heuristic or trial-and-error methods. By explicitly formulating the problem as an MDP, we open pathways to:

- Automate circuit design based on desired frequency characteristics
- Explore extensions into more complex component libraries and non-passive circuits
- Incorporate more advanced RL algorithms such as Deep Q-Networks or Policy Gradient Methods

Future work includes:

- Defining a **reward function** that quantitatively measures the match between the synthesized and target responses
- Implementing and training RL agents to validate the feasibility of the approach
- Extending the framework to handle **noisy** or **approximate** target responses

- Bijeet Basak (IMT2022510)

  Lokesh Aravapalli (IMT2022577)

  Yash Sengupta (IMT2022532)
